MD-Tree: A search-optimized Index Structure for Flash Devices

Karthik V.
Computer Science Dept.
International Institute of Information Technology
Bangalore, India
karthik.v@iiitb.org

G. N. Srinivasa Prasanna
Computer Science Dept.
International Institute of Information Technology
Bangalore, India
gnsprasanna@iiitb.ac.in

***Abstract*—Flash memory provides significant performance improvement over HDDs, but are confronted with some unique challenges like write endurance and write amplifications. Designing of indexing data structures for flash memory has been an area of interest. In this work, we explore a new perspective in implementing B-trees on flash. Previous works have predominantly focused on improving the insert performance which compromised search performance. We propose ways to improves search performance without compromising insert performance using a novel concept of an optimized metadata to speed up the search performance. New ideas to smartly exploit the incremental 1-to-0 update properties of flash in interesting ways to improve the endurance are explored. Our experiments revealed 60% search response time improvement over the best of the alternative schemes. The problems addressed are also relevant to emerging areas like Non-volatile Memory (NVM).**

Keywords—B-tree, Index Structure, KV-Stores, NVM, Flash Memory

I. INTRODUCTION

B-tree and its variants have proved to be the most popular/efficient tree index structure implementations for HDDs as they can easily accommodate large number of records with a small tree height. Thus, entailing minimal disk accesses for processing lookup and update operations. As flash memory adoption increased due its high performance, databases [1, 2, 3, 8] and embedded applications alike benefitted by persisting the tree index data structures in flash/SSD storage. However, we are posed with certain challenges on flash due to the write asymmetry problem. For handling the insert operation on a B-tree node, the key must be inserted into its sorted position within the node and updated on to the underlying storage. Since, flash does not support in-place update (because 1-to-0 bit transitions are supported but not vice versa [1][13]), the current page containing the node has to be marked invalid. The modified contents must be inserted into a fresh page and written (page re-write) to the flash. The invalidated page will subsequently be reclaimed by an erase operation. It can be noted that page re-writes come with associated overheads which not only reduce the device life but affects the device performance as well. There have been some solutions designed over the FTL to conserve such page re-writes [2, 4].

II. RELATED WORK

A brief review of non-FTL based solutions to conserve page re-writes is captured in this section. They could be broadly categorized into two categories as follows:

(i) **Buffering delta updates**: Instead of performing an expensive re-write of the original flash page on every insert, deltas to the page are stored/appended in a separate location/page on flash [3, 6]. This approach improved the insert performance due to buffering as frequent page re-writes are avoided. When a page needs to be read back to memory (search operation) - the base page, as well as delta pages are retrieved, and the current page is re-constructed by re-playing the deltas on the base page contents. The search performance was found to be poor.

(ii) **Lazy-adaptive delta-update**: In order to address the search latency problem, an efficient online adaptive algorithm was developed [5]. The algorithm takes a calculated decision to either retain the delta buffer in memory or to flush it to the flash page. It compares the cost of performance penalty paid by continuing to search an unsorted buffer on one hand and the cost of a re-writing the entire node contents into a fresh flash page on the other hand. Gradually, as the buffer gets filled up, the search performance starts to take a hit (because the unsorted buffer has to be invariably searched in a linear fashion). To overcome this, buffers are emptied (as determined by the online algorithm) into the leaf nodes and re-written in a sorted manner. This essentially adds to the write amplification and write endurance problem which counters the original objctive. The frequency of the re-write is controlled by the workload type – e.g. as the workload become search-intensive, the frequency of page re-write increases. Though the idea strived to improve the search performance, while doing so it compromised on the write endurance which is the fundamental objective behind external memory algorithms for flash.

A. Motivation and novelty

Our goal is to balance both these conflicting requirements i.e. conserving page re-writes and retaining the search performance. We come up with a new indexing scheme called the **MD-tree** (MetaData) for flash. The novelty of the idea is in three fronts: (a) Concept of **balanced partitioning [20]** is employed to divide the data cleverly

into smaller subsets and then sort theses using flash-friendly structures like unary counters (if required). (b) proposes an **optimized metadata** that is attached to the B-tree. At any point in time, using the metadata in conjunction with the B-tree contents, we can quickly re-construct the B-tree in its pristine/sorted form. Therefore, the performance will be on par with conventional B-tree as we deal with **sorted buffers** (and not unsorted buffers in prior art) and (c) The tree and the metadata are constructed/updated by appending contents (and not overwriting) and involving 1-to-0 transitions (i.e. **partial programming**) and this greatly helps to contain the write endurance/amplification effectively.

III. MD-TREE

The MD-tree structure is similar to a B-tree, but with a new way to organize the leaf nodes (here the large majority of changes happen) in a flash-friendly manner (i.e. entail only 1-to-0 transitions). We assume a leaf node can hold L entries, with the requirement that leaf nodes be at least half full [1] and sorted. As new keys are inserted, if we continue to store them in their sorted positions (like done on HDD), it results in a page invalidation on each insertion. Proposed idea tries to solve this problem by building a metadata in a simple and optimal manner. The generic model of the MD-tree leaf node is illustrated in Figure 1.

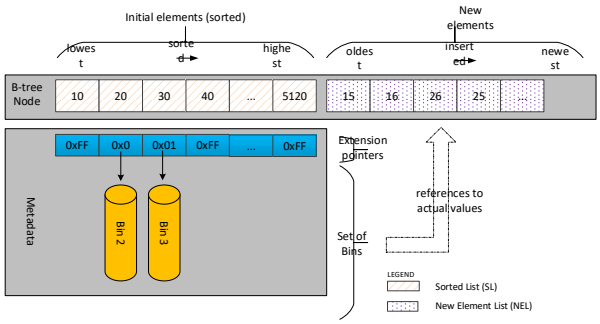


Figure 1 MD-tree leaf node.

A. Data Structures

1. **Sorted List (SL)**: The leaf node portion will be initially half-filled (at least) and sorted. In Fig. 1, initially we had 512 keys and they are sorted/stored then as indicated.

2. **New Element List (NEL)**: As new keys are inserted, they are appended into the node and not placed in their respective sorted positions like done in a conventional B-tree. So, the entries in this portion are not sorted. In the example in Fig. 1, keys 15,16, 26 and 25 have arrived in that order.

3. **MetaData (MD)**: In a conventional B-tree, the keys (initial and new) are always held in sorted order. However, due to reasons explained previously, maintaining the entire node in sorted order is not feasible on flash as it requires performing costly page re-write operations. The obvious result of this is a negative impact on the key search performance. To tackle this problem, the MD is added. This consists of additional data to store offsets to keys in NEL in sorted order. The unique property about MD is that it is updated in a flash-friendly manner. At the time of creation (like any other byte on flash) the MD contents will be initialized to 0xFF (erased state) and further updated via 1-to-0 transitions only. The metadata portion consists of two components:

a) Extension Pointers (EP): The metadata is attached to the SL using EP. For each key in the SL, there is an EP associated. The pointer points to a logical container called a bin.

b) Sorted Bins (SB): In the current example we have chosen $\frac{L}{2}+1$ bins. So, each consecutive key in SL demarcates the bin ranges. If we chose $\frac{L}{4}+1$ bins, then every two consecutive keys in SL demarcate the bin ranges and so on. Each bin contains references to keys in the NEL. Referring to Fig. 1, bin 2 has reference to all new keys in NEL whose value is > 10 and ≤ 20. The bin internally contains a linked list of node(s) (Figure 2). Note: we refer to these nodes as Live Nodes (LN) so that it is not confused with B-tree nodes. Each individual LN has references to max K keys from the NEL of the B-tree. The K keys need not be contiguous in NEL. These references are stored within the LN in a sorted order by using unary counters. Note that the bins are dynamically created from the metadata space and attached to the respective bin range (indicated by the key in SL) by updating the EP with the LN ID.

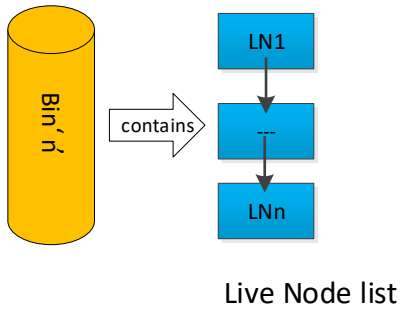


Figure 2 Anatomy a sorted bin

The constituents of a LN are explained below:

LN IDs: LN is uniquely identified by an ID called 'Node ID'. Since, the SB contains a singly lined list of LN(s), each LN points to the next LN via the attribute 'Next Node ID'. The Node ID is a D bits field. The tail LN will have the Next Node ID value as 0xFF. When a LN has to be extended, just 1-to-0 transitions are sufficient (in general the metadata portion is initialized with 0xFFs).

Append-only offset array (AO): Here, we store the offset (not the key itself) of the entries on the NEL that are fall within the respective bin. The AO can hold up to K entries.

Sorted offset array on AO array (SO-AO): In case, we have more entries within AO it would be of interest to further improve the search performance by building sorting information about the AO within the LN. We sort the keys corresponding to the offset entries in the AO array and the sorted offsets are stored separately in an array called SO-AO. This is typically an array of Unary Counters (UC). The counter values represent offsets to the corresponding entries in the AO array. As we know UC can be incremented in-place via 1-to-0 transitions. The size of each counter is C bits and the size of this array is S entries (and S may or may not be equal to K).

Let's insert key 14 into NEL (of Fig. 1). Fig. 3 captures the representation of the respective bins. The Node 0x0 has 3 entries and say we decided that it is appropriate to attach SO-AO to store the sorting information. The first entry in the SO-AO should contain the offset of the entry in AO, which point to smallest key value indicated in NEL for the entries in that AO. The smallest entry is key value 14. Its offset/position on the AO list is index 2. So, the first entry in SO-AO reads 1100b (which is value 2). Similarly, the next higher value is key value 15 and its offset/position on the AO list is 0. Hence, the second entry in the SO-AO reads 1111b (which is value 0). On the same lines, the contents of node ID 0x01 could also be explained.

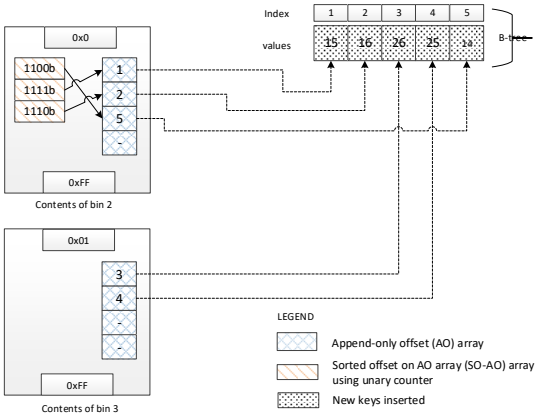


Figure 3 SO-AO

B. Operations on MD-tree

The typical operations performed on a B-tree can be done on MD-tree. Few of them are briefly described below:

Insert: The new entry is appended to the NEL Subsequently the bin in metadata is updated after computing the sorting information. This has two steps:

1) Determination of the associated SB: The key is looked-up in SL using binary search. If the search is unsuccessful, we would have implicitly determined the bin into which the key would belong.

2) **Determination of the non-empty** LN: We further traverse the linked list of LNs within the bin and there are three cases that could arise and handled as follows:

a) Bin is empty: A new LN is created in the bin and the offset of the key on the NEL is inserted into the node's AO. The corresponding EP in SL is updates with the new LN's ID.

b) Tail LN in the bin is filled up (i.e. has 'K' elements): A new LN is created and AO updated like in the previous case. The newly created LN's ID is updated in the tail LN's next nod ID field.

c) Tail LN in the bin is not filled up: The offset of the key is inserted into the AO. If the SO-AO is present in LN, then the counters must be recomputed and updated in SO_AO to reflect the latest sorting information of the LN.

Search: The key is first searched in SL using binary search. If it's unsuccessful, we continue searching within the associated bin. Typically, the bin should have few keys (due to balanced partitioning) which can be quickly using linear search. However, if that is not the case, we search within each LN(s) of the bin sequentially. This can be done quickly using a binary search with the help of the SO-AO. Thus, the search is very efficient as we don't need to deal with large unsorted buffers which was the case with prior art.

Delete: There would be bitmask with the bit positions corresponding to deleted keys offsets on the node [1]. The respective bits are set to indicate key deletions. The bitmask/deleted keys would get purged at the next opportunity of node re-write.

C. SO-AO Implementation

The SO-AO is modelled as a 1-D array of counter values. There are at most K valid counter values within SO-AO and they denote the order/rank of the keys represented by AO. We need to determine the max entries possible in SO-AO for a given length of AO i.e. K. The max size of SO-AO will occur when max INVALID entries are generated within SO-AO. This will occur when we insert x_1 keys in ascending order and x_2 keys with values less than any of the existing keys in SO-AO (Appendix A for proof). The count of invalid counters generated is given by: $(x_1 - 1) * x_2$. With $x_1 + x_2 = K$, we can easily determine:

$$\max_{0<x<K} \sum(invalid\ counters) = \frac{K^2}{4} - \frac{K}{2} \quad \text{for even } K,$$

$$\max_{0<x<K} \sum(invalid\ counters) = \frac{K^2}{4} - \frac{K}{2} + 1/4 \quad \text{for odd } K \quad (1)$$

D. Asymptotic Complexity

Search: In the proposed model there are two levels of search. First, we do a binary search on SL, followed by binary search on the LN. Hence, the worst-case running time cost would be

$$O(log\left(\frac{L}{2}\right) + n * log(K)) \quad (2)$$

Due to the divide-and-conquer strategy, there would be one or two LN(s) present within a bin. In the exceptional case, if there is a build-up of LNs within a single bin (bin overload situation and indicated by $n > T$, where T is a pre-defined threshold), the typical multi-way merge operations could be used to form one larger LN. Post such mitigation, the running cost should be restored to the typical

case. On the other hand, for comparison, the running cost for the B-tree could be expressed as $O\left(log\left(\frac{L}{2}+x\right)\right)$, where x is the total number of new inserts.

Insert: Using Steps 1& 2 mentioned under the insert operation. The insert cost in our proposal would be:

$$O\ (log(L/2) + \mathrm{n} + (\log(K) + S)) \quad (3)$$

The corresponding cost for B-tree would be: $O\left(log\left(\frac{L}{2}+x\right)+\frac{L}{2}+x\right)$. This has the additional step of shifting the keys to create space for the new insertion. We can notice Eqns 2 & 3 are comparable with their B-tree counter parts (and outperform prior art [1]) which is one of the main objectives of this work.

E. Data Partitioning Perspective

It would be of interest to investigate the relative sizes of resulting bins/buckets created within the MD-tree. If the bins are balanced [20], the size of such a portioned would be provided by $count(b_j) = \frac{N}{b}, for\ 1 \le j \le b$ where b_j, N, b indicate j^{th} bin, no of entries in NEL and no of bins respectively. This evaluates to $(\frac{L}{2})/(\frac{L}{2}+1) = 1$ (approx.) in our example. With have one entry with the bin/partition range, from both SL and NEL. Equivalently stated as the probability distribution of SL and NEL are identical (or both sets would have the same mean/standard deviation). Alternatively, the above can also be state in terms of probability of a new key falling into any bin is equal.

$$prob(b_i) = prob(b_{i+1}) = \frac{1}{512}, \text{ for } i = 1\ to\ 513$$

Essentially, we could view the partitioning paradigm as below scenarios at the two far extremes:

Scenario 1: The distribution of NEL follows (i.e. w.r.t mean and standard deviation) the distribution of the SL

Scenario 2: The distribution of NEL does not follow the distribution of the SL.

F. MetaData Memory Size Considerations

In order to conserve the metadata footprint, our idea provides flexibility in configuring/optimizing the size of metadata based on the implementation requirements. The metadata size can be expressed as: $(M * N) + (\frac{L}{2} * p)$. The symbols used are, M: Size(bytes) of the LN, N: Number of LNs, P: Size(bytes) of EP. For a leaf node with 1024 entries and each entry of size 16 bytes, the metadata-to-data overhead would evaluate as: $[(\mathrm{M} * \mathrm{N}) + (256 * 1 + 256 * 2)]/\ 16\mathrm{K}$. As we can see, the size and count of the LN is critical factor in determining the trade-off factor. Our idea provides flexibility in choosing an optimal value for the respective implementations based on the *a priori* distribution information of the newly arriving keys. The value M is a modelled as a function of the number of entries (K) that can be accommodated in it. A lower value for M reduces the search performance and vice versa. The max value of N could be $L/2$. In order to reduce the metadata-to-data overhead, the value of N is chosen lesser than $L/2$, say $N = L/8$. After N new arrivals, the leaf node is purged and we start all over again. This re-cycle would entail one additional page re-write but in return significantly helps to optimize metadata space.

G. Write Endurance

With the conventional implementation of the B-tree used on a flash memory, there will a need for a page re-write after every key insertion in the tree. Our idea reduces it to just **four,** because of two reasons. First, at the algorithm/data structure level by conceptualizing a metadata that can be incrementally updated and second at the physical/device level by making use of the partial programming characteristics.

There are raw flash implementation which support a concept of partial programming i.e. a page could be programmed in parts. So, a flash of size 2K can be programmed separately in chunks of 512 bytes. This concept can be exploited if the data stored in the page is structured in a manner that it can be updated by appending it in chunks. As observed, the NEL or metadata sections like AO work on the principle of appending to an existing list rather than overrating previous values.

As mentioned in [11] partial programming also allows writes to a page, as long as they only involve 1-to-0 transitions. We can notice that the data structures within the metadata like UCs etc. exploit this aspect very well while building the sorting information for the node. The current work is targeted towards devices providing such flash-friendly interfaces or abstractions. Newer technologies on the horizon like Non-Volatile Byte-Addressable memory (NVBM) are good candidates due to their byte-addressable nature.

IV. EXPERIMENTAL EVALUATION

For the implementation a suitable flash simulator [10] was chosen. Discrete objective measures as below were defined and used in order to measure/compare the effectiveness of the algorithms:

- **No of comparisons**: Every key comparison is tracked as a counter which helps to get an idea of the computational complexity of the algorithm,
- **Response time**: This is the end-to-end time duration for the algorithm to complete the tree operations like search, insert.
- **No of page re-writes:** The flash page size was configured to accommodate a leaf node and the no of page invalidations occurring was tracked per leaf node.
- **Metadata size**: This can be statically measured.

In this study, we chose three algorithms for comparative evaluation *viz* the conventional B+-tree, the algorithm

proposed under EWOM model [1] (as it could also be perceived to have a metadata approach) and the MD-tree. The MetaData/page size ratio for the three methods were 0, 12.89% and 12.52% respectively.

A. Probabilistic treatment

Considering an instance of the MD-tree leaf node from a statistical point of view, the SL+NEL represents the actual population and the SL represents the actual samples. For $L = 1024$, sample size will be 512. As captured in section in 2.6, the behavior of the proposed idea is studies by experimenting with different statistical combinations of NEL w.r.t SL. For instance, we chose the [mean, standard deviation] tuple for SL as [500,150] and created three configurations by varying the corresponding tuples for NEL as [500,150], [1500,150] and [800,150] (called config. 1, 2 and 3 respectively), Configs 1 & 2 represent scenarios 1 & 2 respectively. While config. 3 could be viewed as an in-between scenario. The search and insert performance metrics are presented in figures 4 & 5 respectively. Because of the robustness built into the idea, the search performance remained consistent across the three configs in spite of the extreme variations of the input distributions. On 2nd config, there was a bin overload situation observed i.e. more than T=8, nodes within a bin. The algorithm detected a bin overload condition as expected and resorted to a one additional purge/page re-write operation and recovered from overload

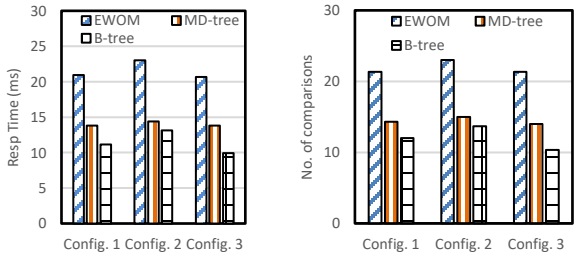


Figure 4: Search workload results

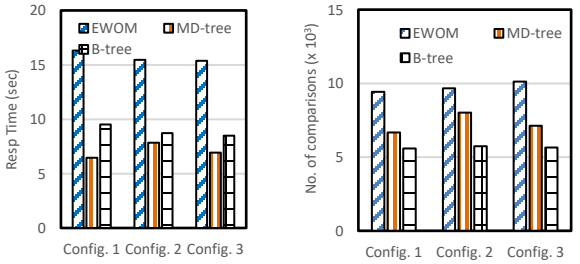


Figure 5: Insert workload results

Tree level operations

A B+-tree was constructed with leaf nodes of different types (i.e. proposed method, EWOM method and a traditional method). We tested with B+-tree with approximately 1 million keys totally (1024 leaf nodes, 1023 keys per leaf node). The tree was subjected to TPC-C mixed workload comprising of insert and search operations in equal ratios with total 50K queries fired on the tree. The cumulative results for each operation captured separately are presented in figures 6 & 7. There is a 60% improvement in search performance which is significant. Additionally, the while doing so did not compromise on the insert performance. The MD-tree could be perceived to provide search performance on par with B-tree which was desired.

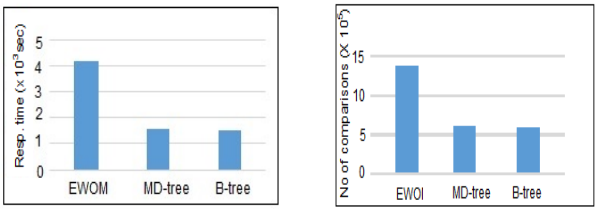


Figure 6: Search workload with large dataset

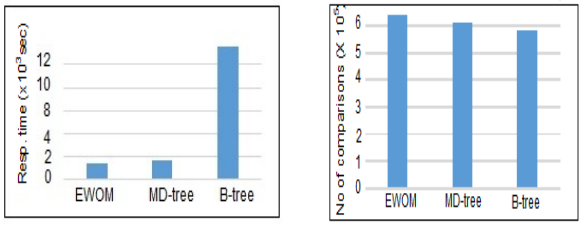


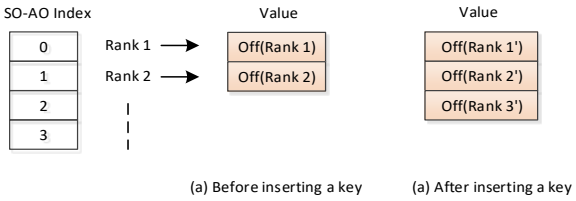
Figure 7: Insert workload with large dataset

Page re-write behavior: The relative ratio of no of page re-writes that occurred per leaf node over $'x'$ insert operations with EWOM, MD-tree and B-tree were: $0, \frac{x}{128}$ and x respectively. We did some experiments/estimations with the state-of-the art, LA-tree and found its performance comparable with EWOM model, but the page re-write was significantly higher:$x/4$.

V. CONCLUSIONS

In this paper, a new way to organize B-tree for fast access on flash memory is presented. The initially populated keys of the node serve to partition the node into equal-sized bins which hold the newly arriving keys. As the node grows, the keys within a bin could be optionally attached with a sorting information. Along with the keys, a minimal metadata is stored in the node to aid in inserting and searching keys. The operations on the node are managed in a flash-friendly manner. The results indicated that MD-tree outperformed the prior art in the search category with 60% performance improvement and was competitive (+4%) in the insert category. The MD-tree brought down the page re-write (i.e. write endurance) frequency by 99.3%. The MD-tree is search-efficient, simple-to-implement & a generic technique which can be deployed on raw flash/NVM.

VI. APPENDIX A



Referring to the above figure, when a new key is inserted into a LN, one of the below outcomes can occur w.r.t to a given SO-AO counter index value:

1) The value remains same i.e. $off_{prev}(rank\ r) = off_{cur}(rank\ r)$.
2) The value changes i.e. $off_{prev}(rank\ r) \neq off_{cur}(rank\ r)$.

SO-AO index entry 0 logically indicates the lowest key value (i.e. rank 1). However, the actual value at index entry 0 represented as $off(rank\ 1)$ indicates the index/offset on the AO entry corresponding to the actual key (inserted into NEL) and lowest value within that LN.

There are two possible outcomes under this case:

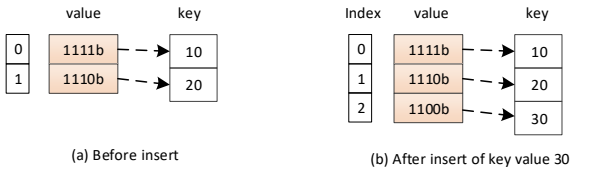
a) The key at rank r previously lost its position to a succeeding key i.e. $off_{prev}(rank\ r) < off_{cur}(rank\ r)$.
b) The key at rank r previously lost its position to a preceding key i.e. $off_{prev}(rank\ r) > off_{cur}(rank\ r)$.

It's interesting to note that the counter value would get invalidated under condition 2-b. The above cases can be intuitively summarized as follows:

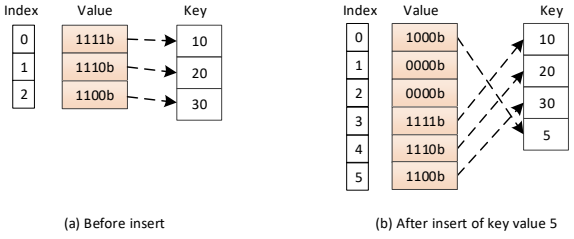
i) We have an opportunity to generate invalid counter entry(s), if there exists key sequence(s) that have arrived in ascending order. In other words, inserting key(s) that that grows the subset of keys in ascending order creates the ground for this.
ii) The above opportunity is exploited by inserting a new key value such which is at least lesser than the highest two elements of such an ascending key sequence(s).

The case i) is referred to as '**Injecting'** a vulnerability (**I**) and case ii) is referred to as '**Exploiting**' the existing vulnerability (**E**). The outcome of an Exploit operation is creation of invalid counter value. A key could be inserted, in one of the below modes.

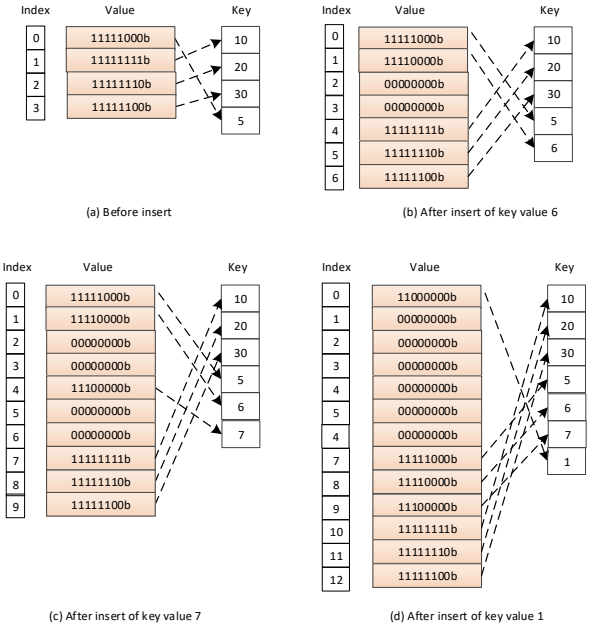
Mode 1: Inject only: In this mode we insert a key such that it creates or appends to a sequence of ascending key values



Mode 2: Exploit only: In this mode a key is inserted which results in generation of new invalid counter value(s).



Mode 3: Exploit and Inject: This mode is a simultaneous combination of mode 1 and 2.



There are couple of logically obvious paths that one could expect to take in order to maximize (E) during the course of insertions.

1. Insert $\boldsymbol{x_1}$ keys i.e. mode 1 and then insert $\boldsymbol{x_2}$ keys in mode 2; Insert $\boldsymbol{x_3}$ keys i.e. mode 1 and then insert $\boldsymbol{x_4}$ keys in mode 2 and so on. $\boldsymbol{x_1 + x_2 + x_3 + x_4 + \cdots = K}$. So, the count of invalid counters would be: $\boldsymbol{(x_1 - 1) * x_2 + (x_3 - 1) * x_4}$ ..
2. Insert $\boldsymbol{x_1}$ keys i.e. mode 1 and then insert $\boldsymbol{x_2}$ keys in mode 3; Insert $\boldsymbol{x_3}$ keys in mode 3 and then Insert $\boldsymbol{x_4}$ keys in mode 3 and so on. $\boldsymbol{x_1 + x_2 + x_3 + x_4 + \cdots = K}$. So, the count of invalid counters would be $\boldsymbol{[(x_1 - 1) * x_2] + [(x_1 - 1) * x_3 + (x_2 - 1) * x_3] + [(x_1 - 1) * x_4 + (x_2 - 1) * x_4 + (x_3 - 1) * x_4]}$+ ..

So, essentially the proposition narrow downs to choosing mode 2 over mode 3 or vice versa. Looking back at the illustrations in Mode 3), while we are inserting keys (keys 5,6 and 7) in order to exploit an existing vulnerability set (10,20 and 30), we have also seeded a new vulnerability. So, there are multiple vulnerability sets (Set 1 has keys 10, 20 and 30 while set 2 has 5, 6 and 7) in the AO. As we insert the next key, going by the greedy approach, one would expect to exploit both the vulnerabilities sets to maximize (E). But, as could be seen from the Mode 3 illustration, we are able to exploit only one vulnerability set (insertion of key value 1, could exploit the vulnerability from set 2 and not from set 1). When we have multiple vulnerability sets, we end up exploiting only one vulnerability set due to a "sliding" effect of the vulnerabilities.

Thus, option 1 approach would maximize the (E). Exploring further on these lines, the option 1 proposition could be maximized under the following proposition: Insert $\boldsymbol{x_1}$ keys in mode 1 and then insert $\boldsymbol{x_2}$ keys in mode 2. In other words, $(x_1 - 1) * x_2$. and count of invalid counters generated is given by: $(x_1 - 1) * x_2$.

VII. REFERENCES

[1] Kenneth A. Ross. Modeling the performance of algorithms on flash memory devices. Proceeding DaMoN '08 Proceedings of the 4th international workshop on Data management on new hardware Pages 11-16 ACM New York, NY, USA 2008.

[2] CHIN-HSIEN WU, TEI-WEI KUO and LI PING CHANG. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. Journal ACM Transactions on Embedded Computing Systems (TECS), Volume 6 Issue 3, July 2007 Article No. 19 ACM New York, NY, USA.

[3] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In SIGMOD Conference, pages 55{66. ACM, 2007.

[4] Y. Li, B. He et al. Tree Indexing on Flash Disks. In ICDE 2009.

[5] D. Agrawal, D. Ganesan, R. Sitaraman, and Y. Diao, "Lazy-Adaptive Tree: An Optimized Index structure for Flash Devices," in Proceedings of the VLDB Endowment, 2009, pp. 361–372.

[6] Hua-Wei Fang, Mi-Yen Yeh, Pei-Lun Suei, and Tei-Wei Kuo. An Adaptive Endurance-Aware B+-Tree for Flash Memory Storage Systems IEEE Trans. Computers 63(11):2661-2673 (2014).

[7] Xiaona Gong, Shuyu Chen, Mingwei Lin, Haozhang Liu. A Write-Optimized B-Tree Layer for NAND Flash Memory. Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference, pages 1-4, 2011.

[8] Luis Cavazos Quero, Young-Sik Lee, and Jin-Soo Kim. Self-sorting SSD: Producing sorted data inside active SSDs. MSST, pages 1-7, IEEE, 2015.

[9] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing, Proceedings of the VLDB Endowment, Vol. 7(14), 2014.

[10] Y. Kim, B. Tauras, A. Gupta and B. Urgaonkar, "FlashSim: A Simulator for NAND Flash-Based Solid-State Drives," 2009 First International Conference on Advances in System Simulation, Porto, 2009, pp. 125-131.

[11] Intel Corp. Understanding the Flash Translation Layer (FTL) Speci_cation, 1998.

[12] Deepak Ajwani, Itay Malinger, Ulrich Meyer and Sivan Toledo. Characterizing the performance of Flash memory storage devices and its impact on algorithm design. Proceeding WEA'08 Proceedings of the 7th international conference on Experimental algorithms Pages 208-219, 2008.

[13] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives", in SYSTOR 2009:The Israeli Experimental Systems Conference, 2009.

[14] Paul England and Marcus Peinado. System and method for implementing a counter, 2006. US Patent Number 7,065,607.

[15] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin and Sang-Won Lee. B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. Journal Proceedings of the VLDB Endowment VLDB Endowment Homepage archive Volume 5 Issue 4, December 2011 Pages 286-297.

[16] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high performance flash disks. In SIGOPS Operating Systems Review, volume 41(2), pages 88{93, 2007.

[17] D. Myers and S. Madden. On the use of NAND flash disks in high performance relational databases. 2007.

18] Suman Nath and Aman Kansal. Flashdb: dynamic self-tuning database for NAND flash. In IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks, pages 410-419, New York, NY, USA, 2007. ACM.

[19] E. Gal and S. Toledo. Algorithms and data structures for flash memories. In ACM Computing Surveys, volume 37, pages 138-163, 2005.

[20] X. Shen and C. Ding. Adaptive data partition for sorting using probability distribution. In Proceedings of International Conference on Parallel Processing, Montreal, Canada, August 2004.

[21] Sang-Won Lee , Bongki Moon , Chanik Park , Jae-Myung Kim , Sang-Woo Kim, A case for flash memory ssd in enterprise database applications, Proceedings of the 2008 ACM SIGMOD international conference on Management of data, June 09-12, 2008, Vancouver, Canada.

[22] Goetz Graefe. Modern B-Tree Techniques. Foundations and Trends in Databases , 3(4):203–402, 2011.

[23] M. Huang, O. Serres, V. K. Narayana, T. El-Ghazawi, and G. Newby, "Efficient cache design for solid-state drives, " in Proc. 7th ACM International Conference on Computing Frontiers (CF'10), May 2010, pp. 41-50.

[24] Jeffrey Scott Vitter, Algorithms and data structures for external memory, Foundations and Trends® in Theoretical Computer Science, v.2 n.4, p.305-474, January 2008.

[25] Dimitris Tsirogiannis , Stavros Harizopoulos , Mehul A. Shah , Janet L. Wiener , Goetz Graefe, Query processing techniques for solid state drives, Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, June 29-July 02, 2009, Providence, Rhode Island, USA.

[26] Ning Zhang , Junichi Tatemura , Jignesh M. Patel , Hakan Hacigümüş, Towards cost-effective storage provisioning for DBMSs, Proceedings of the VLDB Endowment, v.5 n.4, p.274-285, December 2011.

[27] Mustafa Canim , George A. Mihaila , Bishwaranjan Bhattacharjee , Kenneth A. Ross , Christian A. Lang, An object placement advisor for DB2 using solid state storage, Proceedings of the VLDB Endowment, v.2 n.2, August 2009.

[28] Shimin Chen, FlashLogging: exploiting flash devices for synchronous logging performance, Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, June 29-July 02, 2009, Providence, Rhode Island, USA.

[29] Ioannis Koltsidas , Stratis D. Viglas, Data management over flash memory, Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, June 12-16, 2011, Athens, Greece.